DIGITAL SYSTEMS DEVELOPMENT AND TESTING
PRACTICAL WORK 2 REPORT

Ayse Idman,
64047,
idmanayse@gmail.com;

Kaan Ari,
64048,
kaan.ari.tr@gmail.com;

1 Introduction

In this project, we aimed to design a configurable test equipment device for custom circuit that contains 2 74HC153N (Dual 4-input multiplexer with active LOW enable)

Here is the requested circuit:

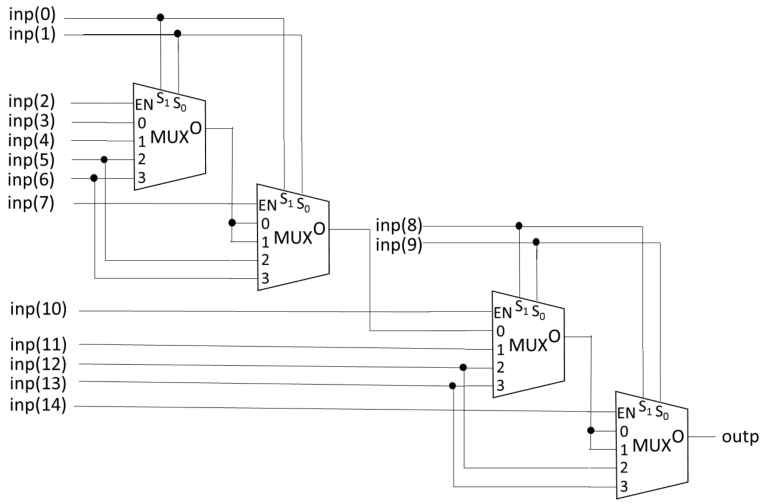


Fig. 1. Provided Circuit Under Test

The requested circuit (Circuit/Unit Under Test) is almost correct. We found that EN inputs of all the muxes are low enable. So that, we slightly changed this configuration in our simulations to obtain true behavior of the system and find good count/signature value.

The improvements of the circuit under test for true simulations can be seen in figure 2 below.

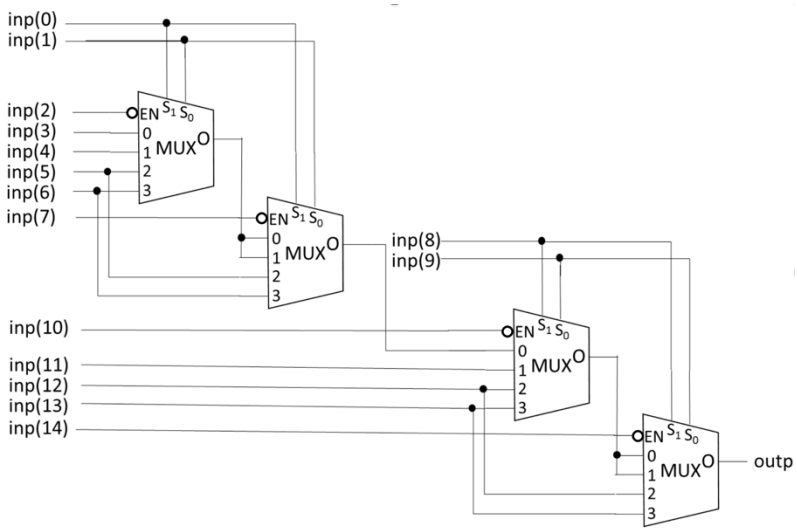


Fig. 2. Corrected Circuit Under Test

We first start with designing 74HC153N module in VHDL. Then, we created another file named as UnitUnderTest.vhd to imitate provided configuration. As it can be seen in figure 3, we connected two 74HC153N modules together. Hence, we imitated 4 muxes in figure 2.

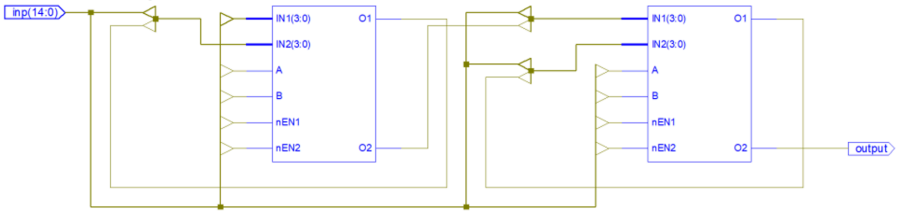


Fig. 3. VHDL Schematic of Circuit Under Test

The general architecture of test equipment has been provided us (figure 4). We generally follow it. However, we wanted to create more configurable and versatile design. To do that, we always use VHDL generic parameters. For example, if we design a counter, we also provide a generic variable to define the number of registers in counter.

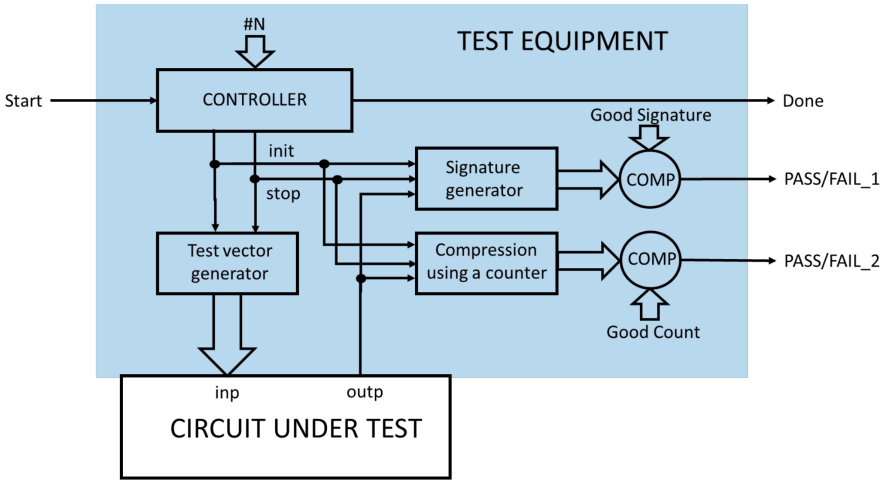


Fig. 4. Provided Test Equipment Design

Here is our final test equipment architecture with improvements:

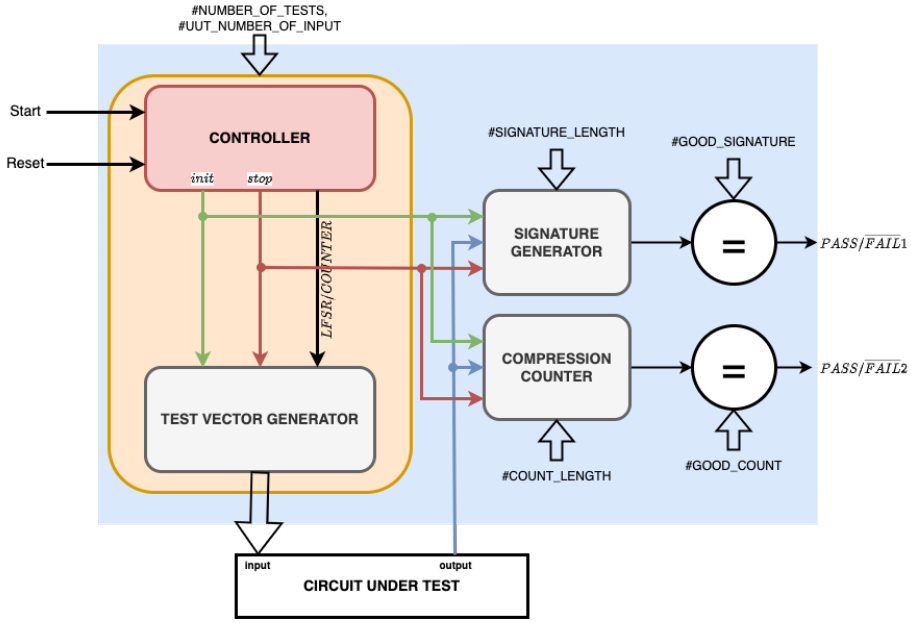


Fig. 5. Improved Test Equipment Design

With improvements, we can easily change configurations. For example, if we want counter in test vector generator, we set #NUMBER_OF_TESTS to $2^{UUT\_NUMBER\_OF\_INPUT}$. If #UUT_NUMBER_OF_INPUT equal to 15 as it is like in our case, we set #NUMBER_OF_TESTS to 32768. Then, LFSR/Counter bit is set to 0 for indication of using counter. If we set #NUMBER_OF_TESTS to another number, controller automatically prefers using LFSR to generate random input test vectors. Also, we designed variable signature and count bit lengths to make test verification adjustable. As a result of these additions, we just use 3 of our final design to generate 3 different systems (Exhaustive Test Equipment (Counter), Random Test Equipmen1 (LFSR with the number of tests is equal to 128), and Random Test Equipmen2 (LFSR with the number of tests is equal to 8192)). It makes our job easy because we only designed the components once and adjust it from the outside to generate different test equipment. Moreover, we also additional reset and clock signals to simulate it in FPGA.

1.1 Exhaustive Test Equipment (Counter)

As we mentioned before, the design of the equipment is same. We will just state its unique specifications.

UUT_INP_N	15
N_TESTS	32768
SIGNATURE_LENGTH	7
GOOD_SIGNATURE	0x0054
COUNT_LENGTH	7
GOOD_COUNT	0x0220

In all three system configurations we used SIGNATURE_LENGTH as 7 because it gives the probability of aliasing is equal to 0.78% (~2^{-7}). This is an acceptable probability for our test equipment, so we used it for all three test systems. However, we change COUNT_LENGTH for each system because we do not want to get aliasing in counter. Since we use edge counter for compression counter, we can get $32768 + 1$ (#N_TESTS+1) maximum. Hence, we used a 16-bit counter for exhaustive test equipment.

Good count and good signature parameters determined from the VHDL simulations.

1.2 Random Test Equipment 1 with 128 Tests (LFSR)

UUT_INP_N	15
N_TESTS	128
SIGNATURE_LENGTH	7
GOOD_SIGNATURE	0x0047
COUNT_LENGTH	8
GOOD_COUNT	0x0016

1.3 Random Test Equipment 2 with 8192 Tests (LFSR)

UUT_INP_N	15
N_TESTS	8192
SIGNATURE_LENGTH	7
GOOD_SIGNATURE	0x0077
COUNT_LENGTH	14
GOOD_COUNT	0x0874

2 Implementations

In this section, we describe how we implement all the subunits of test equipment.

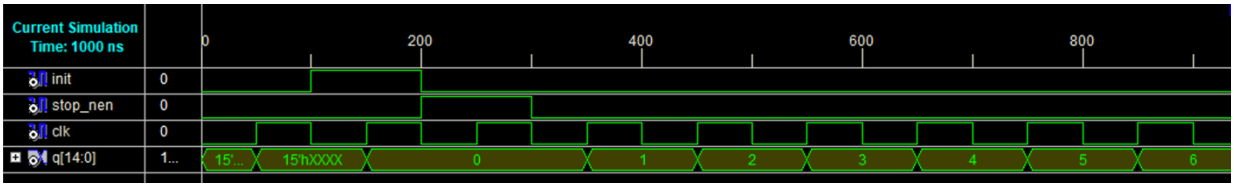
2.1 Generic Counter (Counter.vhd)

We design a generic counter to be used in compression counter and controller. Generic variable N controls the size of the counter.

Here is our counter implementation:

```
entity Counter is
       generic(N : integer := 15);
    Port ( init : in  STD_LOGIC;
           stop_nEN : in  STD_LOGIC;
           CLK : in  STD_LOGIC;
           Q : out  STD_LOGIC_VECTOR (N-1 downto 0));
end Counter;

architecture Behavioral of Counter is
       signal count : unsigned(N-1 downto 0);
begin

       process(CLK)
       begin

              if rising_edge(CLK) then
                     if init = '1' then
                            count <= (others => '0'); -- Synchronous Reset
                     elsif stop_nEN = '0' then
                            count <= count + 1; -- Count
                     end if;
              end if;

       end process;

       Q <= std_logic_vector(count);

end Behavioral;
```

Here is our testbench for counter:



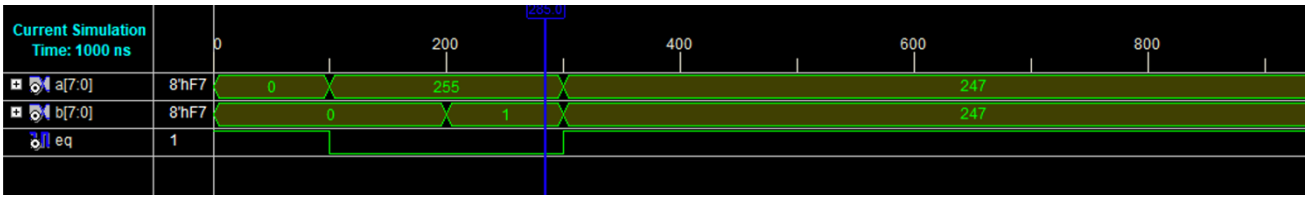
2.2 Generic Comparator (Comparator.vhd)

We design a generic comparator to check pass/fail of the test. Generic variable N controls the input size of the comparator.

```
entity Comparator is
      generic(N : integer := 4);
    Port ( A : in  STD_LOGIC_VECTOR (N-1 downto 0);
           B : in  STD_LOGIC_VECTOR (N-1 downto 0);
           Eq : out  STD_LOGIC);
end Comparator;

architecture Behavioral of Comparator is
begin

process(A, B)
begin
      if A = B then
            Eq <= '1';
      else
            Eq <= '0';
      end if;
end process;

end Behavioral;
```

Here is our counter implementation:

Here is our testbench for comparator:

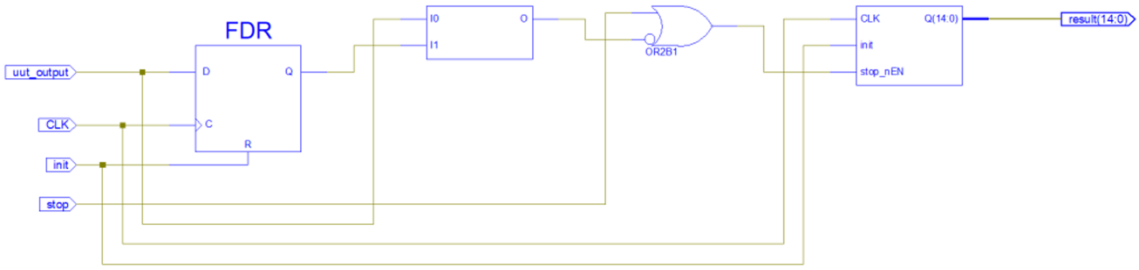


2.3 Generic Compression Counter (CompressionCounter.vhd)

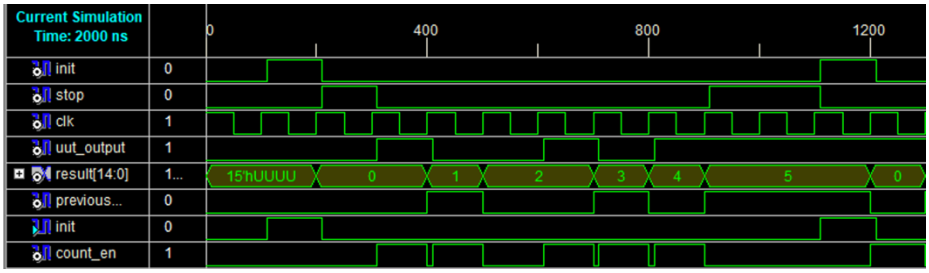
We design a generic compression counter by using existing generic counter. The only difference is we set the enable pin of the counter if there is a change between the past and current input. Hence, we aimed to use transmission compression technique (edge transition).

The implementation code cannot fit that page, but it can be seen in project folder.

Here is our counter implementation:



Here is our testbench for compression counter:



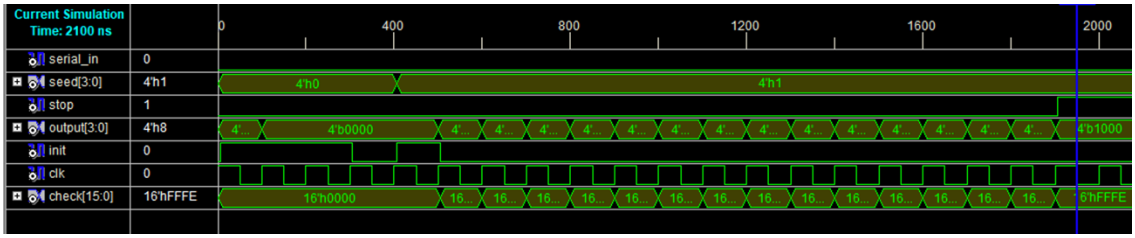
2.4 Lineer Feedback Shift Register (LFSR.vhd)

We design a generic LFSR (up to 15-bit long). To obtain maximal sequence LFSR, we wrote primitive polynomials in a constant lookup table in the code. As a result, we can change the length of the LFSR from 2 to 15. Since 15-bit LFSR is enough for our application, we only searched and inserted primitive polynomials up to 15. Our source was Wikipedia. It provides primitive polynomials up to 24-bit LFSR.
To make reconfigurable LFSR, we changed the XOR tap position by looking lookup table. Whenever our VHDL code synthesis, we get correct tap positions since we have lookup table.
Not but not least, we also add serial input pin to LFSR to use LFSR both in Signature Generator and Test Vector Generator. If serial input is 0, it behaves like autonomous LFSR without serial input.

Here is our LFSR implementation:

```
entity LFSR is
        Generic(SEQ_LENGTH : integer := 15);
    Port ( serial_in : in  STD_LOGIC;
                          init : in STD_LOGIC;
                          Seed : in STD_LOGIC_VECTOR(SEQ_LENGTH-1 downto 0); -- Initial Value
                          stop : in STD_LOGIC;
                          CLK : in STD_LOGIC;
           output : out  STD_LOGIC_VECTOR(SEQ_LENGTH-1 downto 0));
end LFSR;

architecture Behavioral of LFSR is
        constant MAXIMUM_SEQ_LENGTH : integer := 15;
        constant MINIMUM_SEQ_LENGTH : integer := 2;

        type STD_ARRAY is array (MINIMUM_SEQ_LENGTH to MAXIMUM_SEQ_LENGTH) of
STD_LOGIC_VECTOR(MAXIMUM_SEQ_LENGTH-1 downto 0);
        constant TAP_LOOKUP : STD_ARRAY := ("000000000000011",
        "000000000000110","000000000001100",
        "000000000010100","000000000110000",
        "000000001100000","000000010110100",
        "000000100010000","000001001000000",
        "000010100000000","000111000001000",
        "001110010000000","011100000000010",
        "110000000000000");

        signal REG : STD_LOGIC_VECTOR(SEQ_LENGTH-1 downto 0);
        signal SELECTED_TAPS : STD_LOGIC_VECTOR(SEQ_LENGTH-1 downto 0);
        signal FEEDBACK : STD_LOGIC;

        constant TAP_POSITIONS : STD_LOGIC_VECTOR(SEQ_LENGTH-1 downto 0) :=
TAP_LOOKUP(SEQ_LENGTH)(SEQ_LENGTH-1 downto 0);

begin
        process(CLK)
        begin
                if rising_edge(CLK) then
                        if init = '1' then
                                REG <= Seed;
                        elsif stop = '0' then
                          REG(SEQ_LENGTH-1 downto 1) <= REG(SEQ_LENGTH-2 downto 0); -- SHIFTING LEFT
                          REG(0) <= FEEDBACK;
                        end if;

                end if;

        end process;

        SELECTED_TAPS <= TAP_POSITIONS AND REG; -- Selecting Corresponding Tap Positions

        process(SELECTED_TAPS, serial_in)
                variable xor_result : STD_LOGIC;
        begin
                xor_result := SELECTED_TAPS(0);
                for idx in 1 to SEQ_LENGTH-1 loop
                        xor_result := xor_result XOR SELECTED_TAPS(idx);
                end loop;

                FEEDBACK <= xor_result XOR serial_in;

        end process;
        output <= REG;

end Behavioral;
```

Here is our testbench for LFSR:



4-bit long LFSR testbench is shown. However, we tested all the LFSR size from 2 to 14. To do that, we generate a check STD_LOGIC_VECTOR and we set one of its bit corresponding to LFSR output. At the end we must see 0xFFFE in result variable because LFSR must be equal all the possibilities except 0. Also, instead of checking by hand, we add assertion statements in the testbench to see whether all the possible values are reached or not. If there is an error (same number repeated in the sequence twice or an unreached possibility except 0), we would see it at the simulation console.

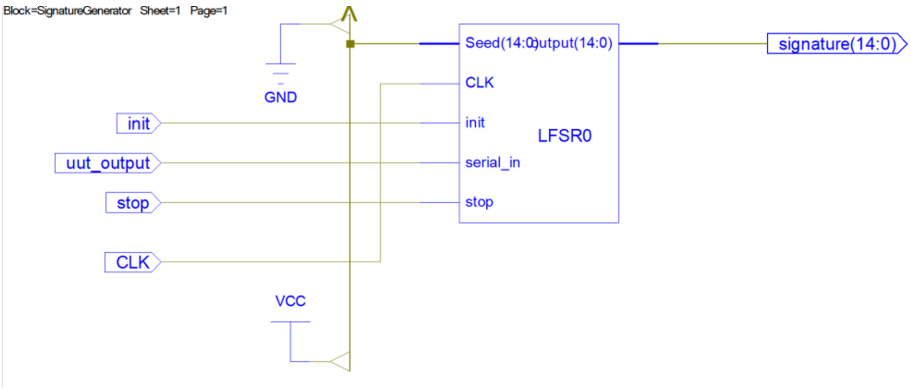
Example of an assertion statement:

```
assert False report "ERROR: NOT ALL THE POSSIBILITIES ACHIEVED!" severity error;
```

2.5 Signature Generator (SignatureGenerator.vhd)

We design a generic signature generator to calculate the signature of the circuit under test. It is basically an LFSR with serial input. Since we designed our LFSR with serial input. We did not do anything special instead of using existing LFSR design. Also, we did not prepare a testbench for Signature Generator because it just contains LFSR and we had already tested our LFSR.
Seed value is selected as “0x0001” for all three systems.

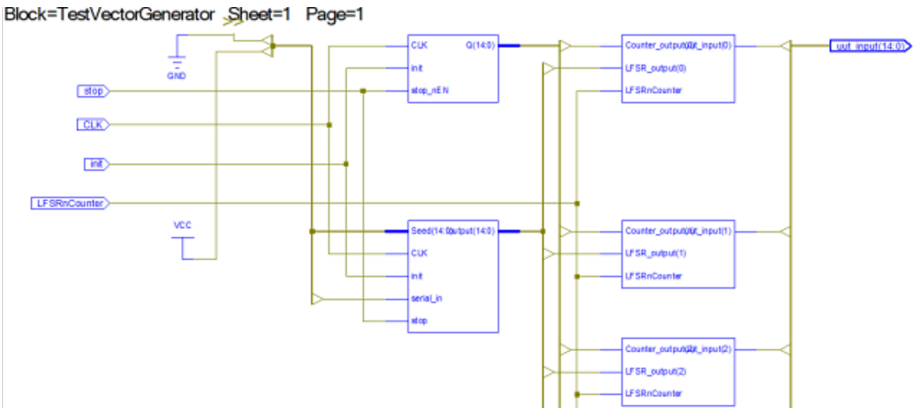
Here is our signature generator implementation:



2.6 Test Vector Generator (TestVectorGenerator.vhd)

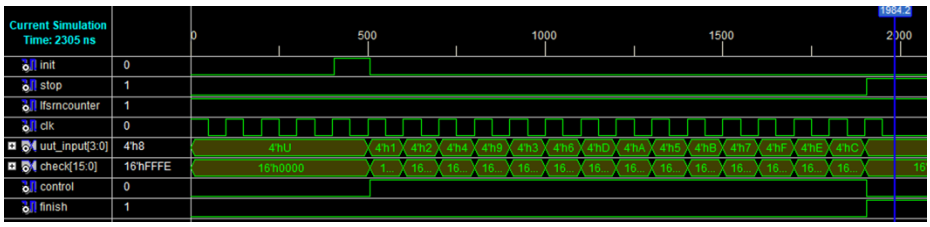
We design a generic test vector generator by using existing generic counter and LFSR. We can state the test vector size and it can create proper size LFSR and counter by using that generic variable. Furthermore, we can choose counter or LFSR as a test vector driver by using LFSRnCounter pin.

Here is our Test Vector Generator implementation:

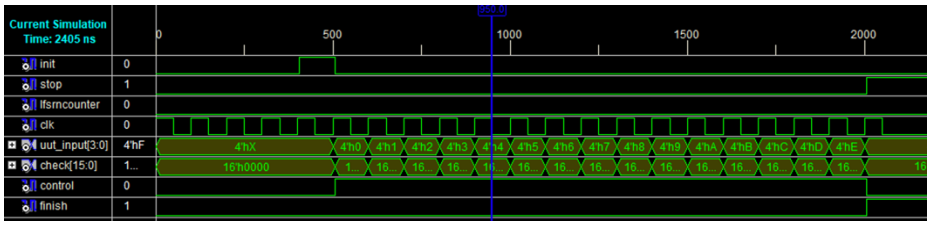


We cropped bottom of the design because right hand side components just represent a 15-bit multiplexer together. Two components in the middle of the design are LFSR and counter. Output of the module can be selected by LFSRnCounter pin. If LFSRnCounter is 0, Counter will drive the output. Otherwise, LFSR will drive the output. We provide tests for 4-bit output because it is hard to put 15-bit result in the report. We also use assertion statements here because it makes easier to simulate longer bit input circuits.

Here is our testbench for Test Vector Generator (LFSR is driving the output):



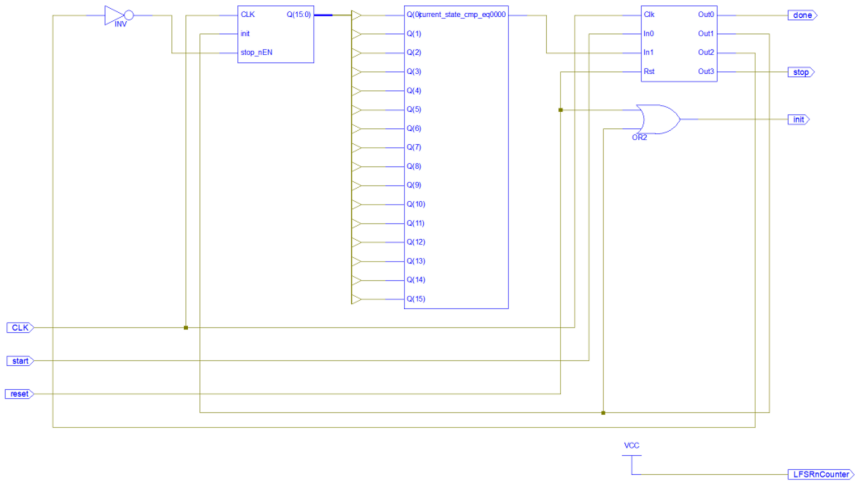
Here is our testbench for Test Vector Generator (Counter is driving the output):



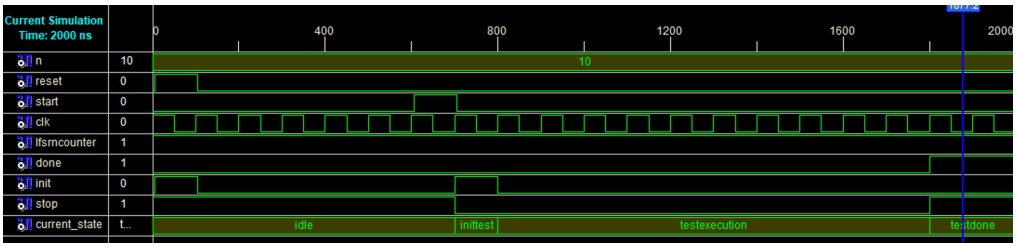
2.7 Controller (Controller.vhd)

We design a generic controller. It basically does synchronization of all the components. We built it as a Finite State Machine (FSM). It has 4 states, Idle, InitTest, TestExecution, and TestDone. When in reset input comes to controller, it returns to idle state. When it is in Idle state, if the start pin is set, then the next state will be InitTest.
InitTest state will last for 1 clock cycle and the next state will be automatically TestExecution. Then the controller start an internal counter and wait it to count up to number of test. After counter reaches the number of test, controller changes its state to testDone state. All the signal assignments are done in each state. As a result, the logic is simplified by using FSM approach.

Here is our Controller implementation:



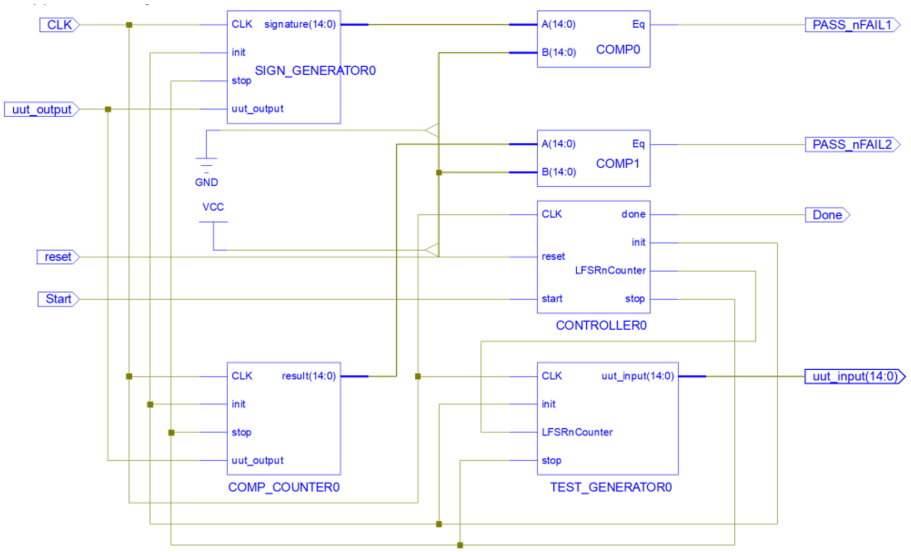
Here is our testbench for Controller (Number of test to be executed is 10):


Current Simulation Time: 2000 ns
0
400
800
1200
1600
2000
n
10
10
reset
0
start
0
clk
0
lfsrncounter
1
done
1
init
0
stop
1
current_state
t...
idle
inittest
testexecution
testdone

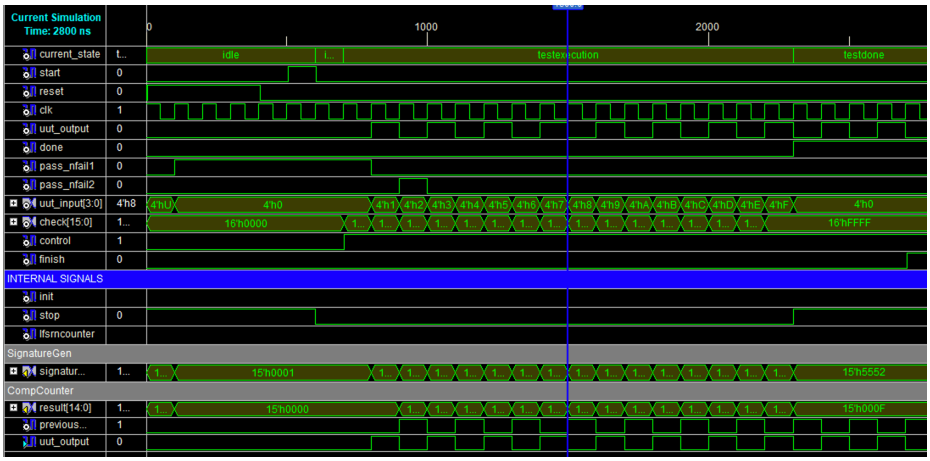
2.7 Test Equipment (TestEquipment.vhd)

After designinga and testing all the submodules, it was easy to build test equipment. We only connected inter connection between modules and wrote a testbench.

Here is our Test Equipment implementation:



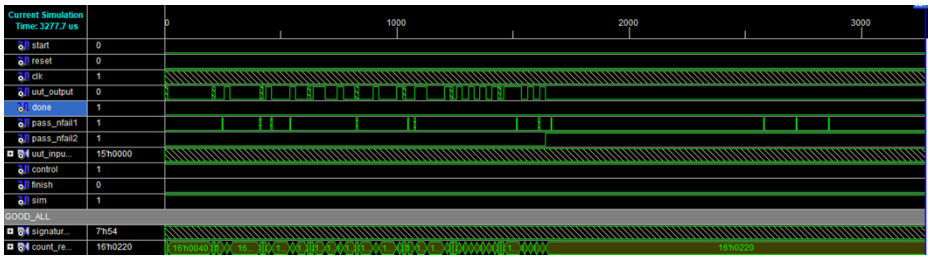
Here is our testbench for Test Equipment without circuit under test (CUT)
Number of tests = 16, UUT_INPUT_N = 4. Hence, counter was driving the output.


Current Simulation Time: 2800 ns
0
1000
2000
current_state
t...
idle
i...
testexecution
testdone
start
0
reset
0
clk
1
uut_output
0
done
0
pass_nfail1
0
pass_nfail2
0
uut_input[3:0]
4'h8
4'hU
4'h0
4'h1
4'h2
4'h3
4'h4
4'h5
4'h6
4'h7
4'h8
4'h9
4'hA
4'hB
4'hC
4'hD
4'hE
4'hF
4'h0
check[15:0]
1...
16'h0000
16'hFFFF
control
1
finish
0
INTERNAL SIGNALS
init
stop
0
lfsrncounter
SignatureGen
signatur...
1...
15'h0001
15'h5552
CompCounter
result[14:0]
1...
15'h0000
15'h000F
previous...
1
uut_output
0

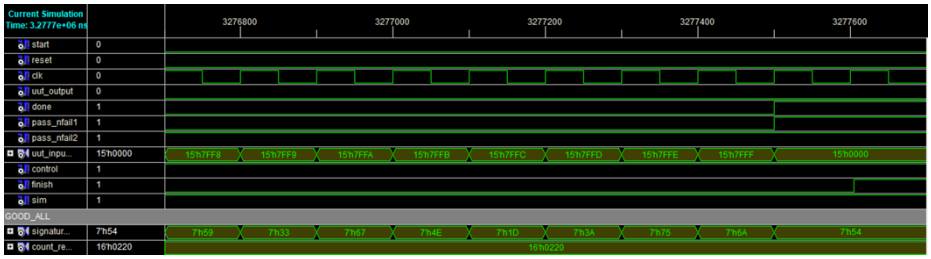
3 Final Testbench (To find good count and signature for all three system configuration)

We designed a testbench and connect the test equipment with circuit under test. Hence, we was able to determine good count, and good signature value for each system.

3.1 Exhaustive Test Equipment (Counter)

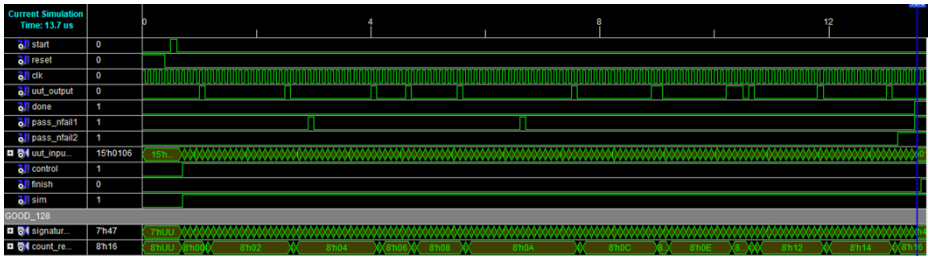


Since it is hard to see last steps, we also captured the screenshot of the simulation end.



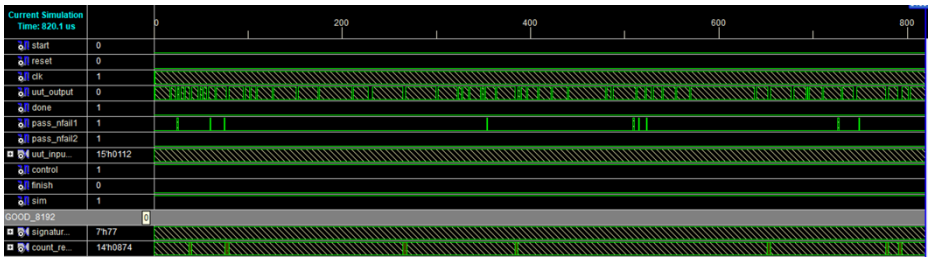
As it can be seen in the simulation, GOOD_SIGNATURE for exhaustive test equipment is 0x0054 and GOOD_COUNT is 0x0220.

3.2 Random Test Equipment 1 with 128 Tests (LFSR)

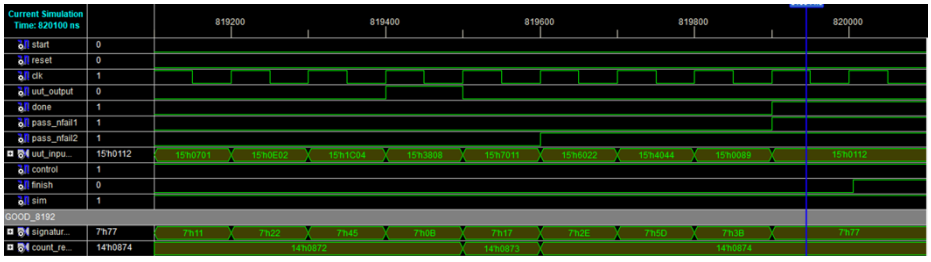


As it can be seen in the simulation, GOOD_SIGNATURE for Random Test Equipment 1 is 0x0074 and GOOD_COUNT is 0x0016.

3.3 Random Test Equipment 2 with 8192 Tests (LFSR)



Since it is hard to see last steps, we also captured the screenshot of the simulation end.



As it can be seen in the simulation, GOOD_SIGNATURE for Random Test Equipment 2 is 0x0077 and GOOD_COUNT is 0x0874.

4 FPGA Implementation

As we said earlier, we put all three system in FPGA. To do that, we add a couple of functionalities also. We used built-in push buttons on the board to stimulate reset and start inputs. We used slide switches to select among three test equipment. Furthermore, we used LEDs to show test equipment selection, to show test results (Done, PASS/FAIL1, PASS/FAIL2 outputs).

We showed all the functions on board in figure 6 below.

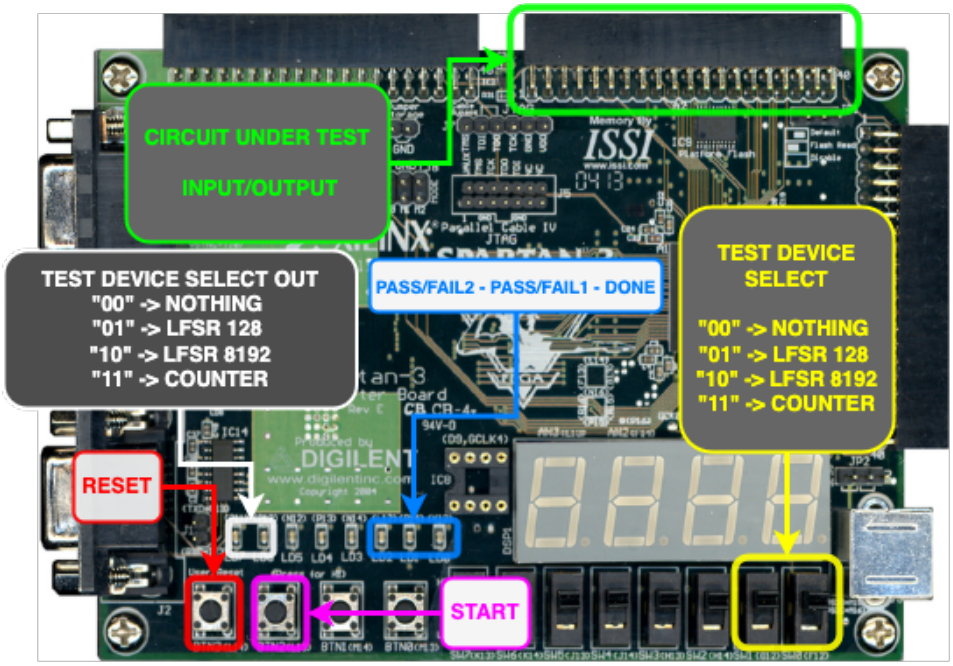


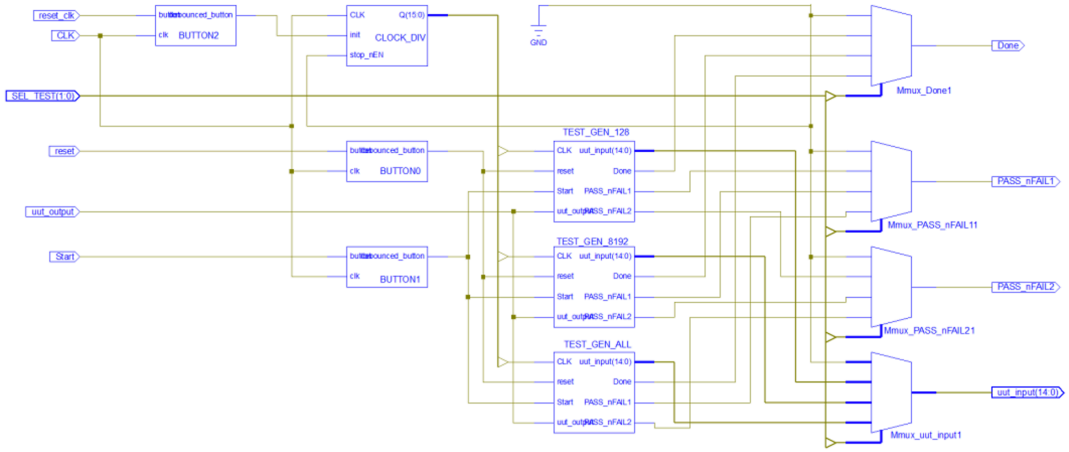
Fig. 6. Usage of FPGA resources

To employ three test equipment, buttons, switches, and CUT outputs, we wrote a wrapper code (FPGA_IMPLEMENT.vhd).

In the wrapper, we inserted debounce logic and a clock divider counter. Debounce logic is necessary because we wanted to be sure there is no unstability in the input signal. Also we added a counter to divide internal clock because the clock frequency is 50 Mhz. In our first try, we got errors due to high speed of clock signal. We thought that

74HC153N is not capable to work under 50 Mhz clock signal. As a result, we add a 16-bit counter to divide clock. It is not recommended to use counter rather than clock divider Ips but we could not manage to use IP creator of Xilinx properly. So that, we used a simple counter to divide clock. After division we assign different clock frequencies for each test system because counter has a longer sequence to test than other two systems. By connecting different clock speeds to each test equipment, we got suitable test execution times.

Here is the design of our FPGA wrapper module:



It consists three debounce logic circuit, 1 clock divider, 3 test equipment and 4 multiplexer. With this final design, we were able to test all three configuration with only one time programming of the FPGA.

5 Real Test Results of the circuit without failures

We connected 2 74HC153N on a breadboard with the proposed configuration. Then we connect its inputs and output to A2 extension adapter. Then, we initiated the test of the circuit without failures.

5.1 Random Test Equipment 1 with 128 Tests (LFSR)

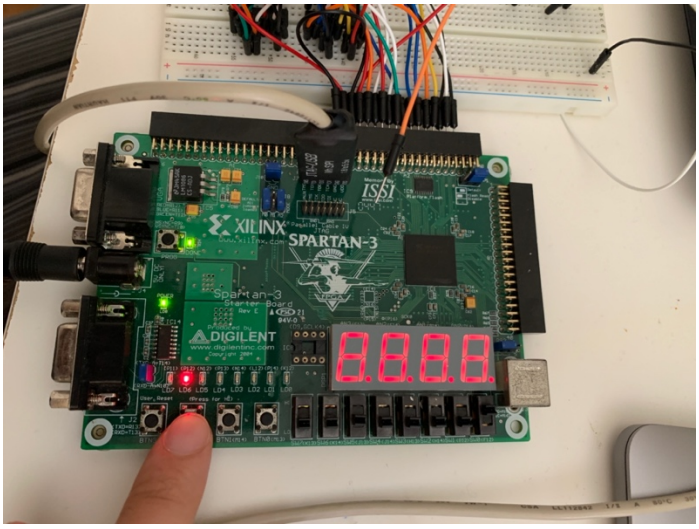


Fig. 7. Before Test (LFSR with 128 tests) – Select Switches were configured as “01”

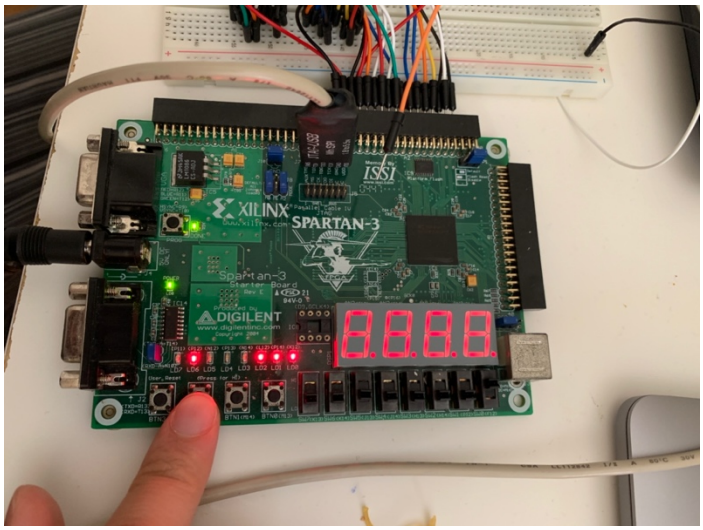


Fig. 8. After Test (LFSR with 128 tests) – Tests passed

5.2 Random Test Equipment 2 with 8192 Tests (LFSR)

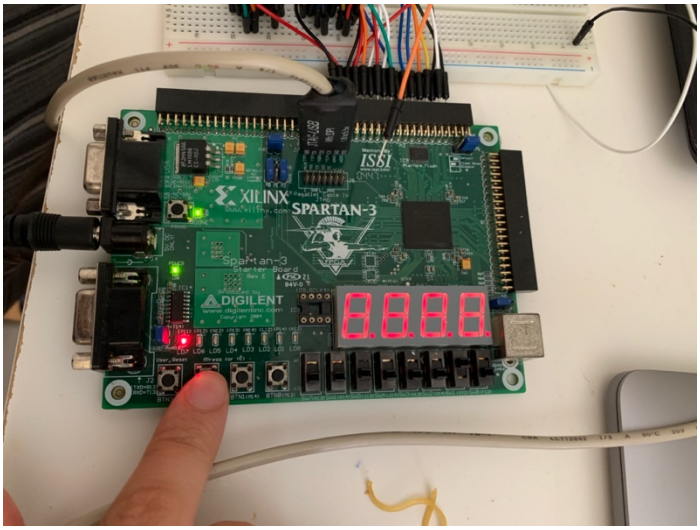


Fig. 9. Before Test (LFSR with 128 tests) – Select Switches were configured as “10”

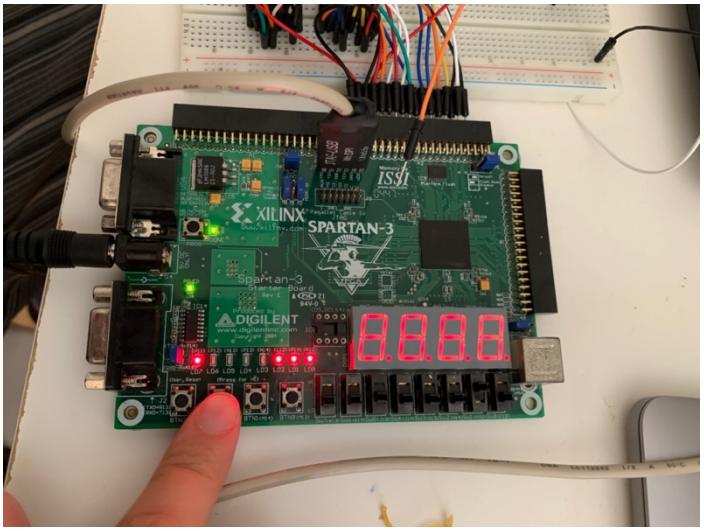


Fig. 10. After Test (LFSR with 8192 tests) – Tests passed

5.3 Exhaustive Test Equipment Test (Counter)

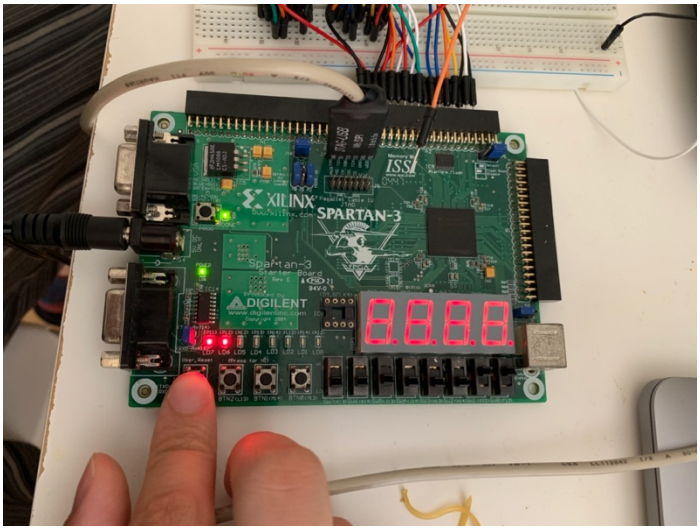


Fig. 11. Before Test (Counter with exhaustive test) – Select Switches were configured as “11”

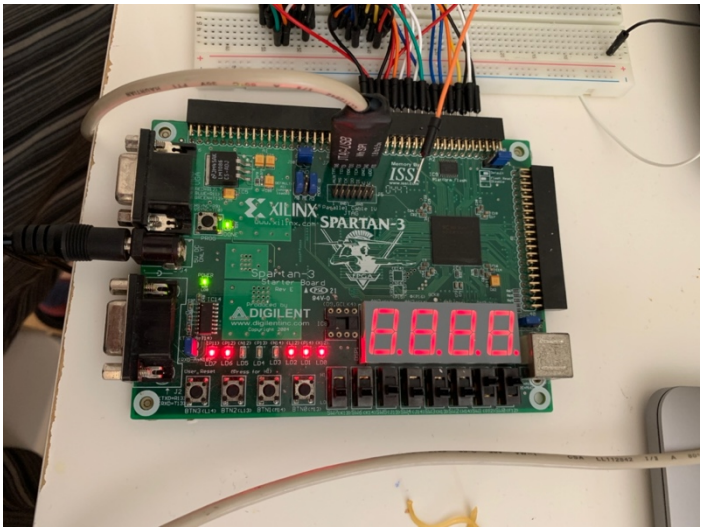


Fig. 12. After Test (Counter with exhaustive test) – Tests passed

6 Real Test Results of the circuit with failures

6.1 Input(0) Stuck At 0

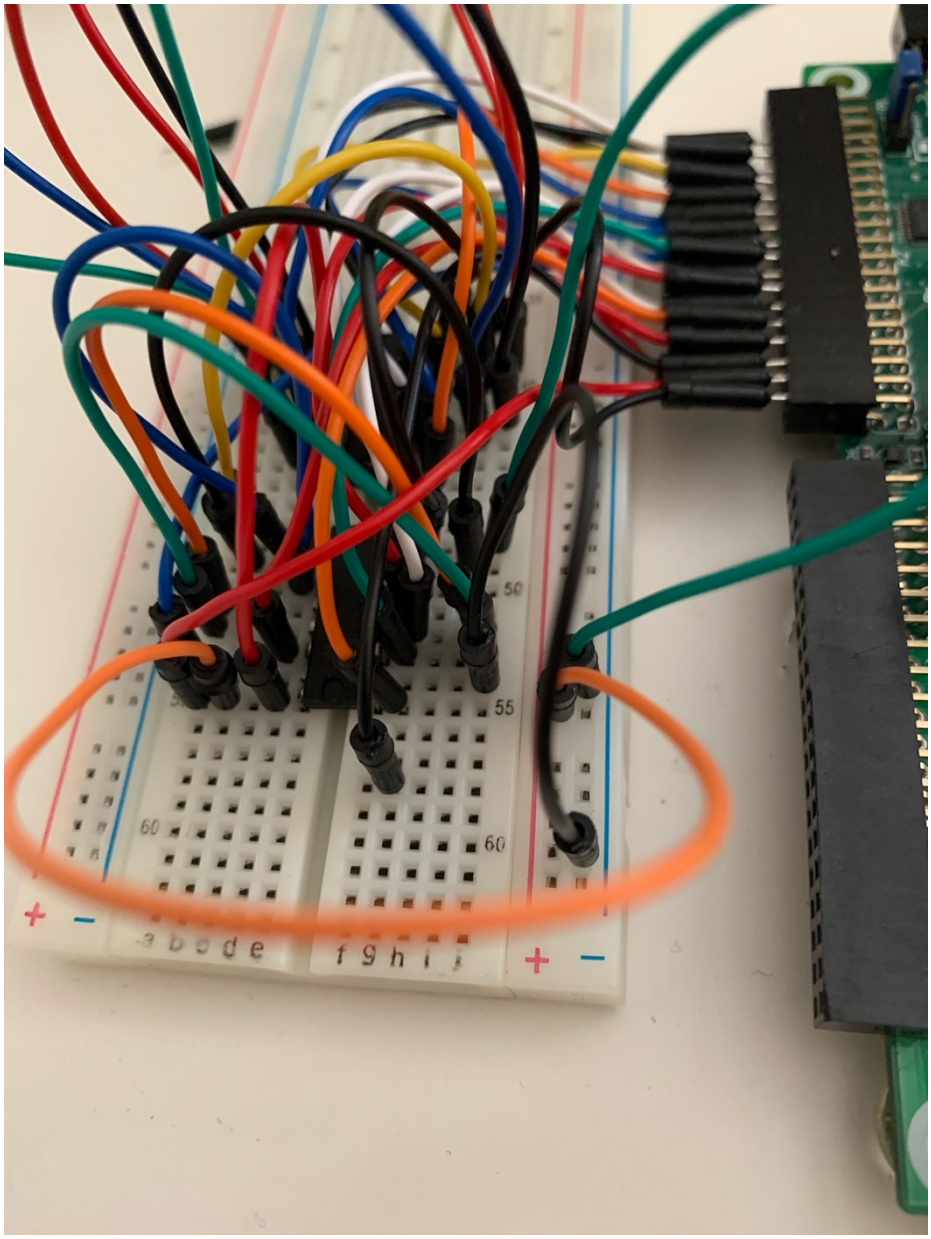


Fig. 13. Inserting stuck at 0 to input(0)

6.1.1 Random Test Equipment 1 with 128 Tests (LFSR)

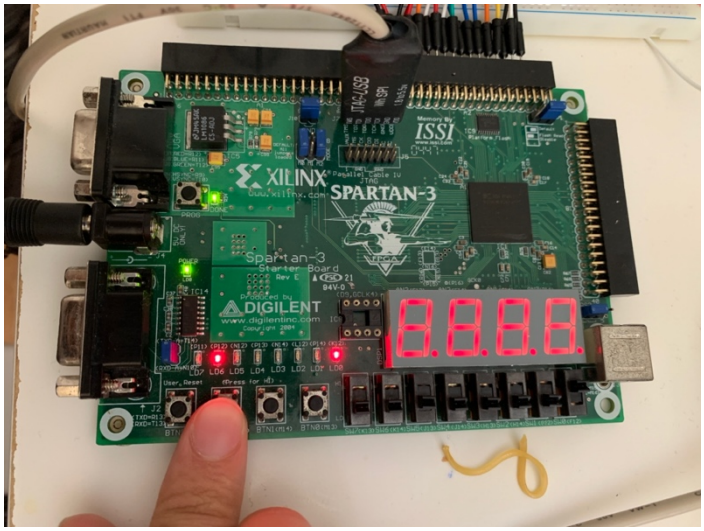


Fig. 14. After Test (LFSR with 128 tests) – All tests failed (as expected)

6.1.2 Random Test Equipment 2 with 8192 Tests (LFSR)

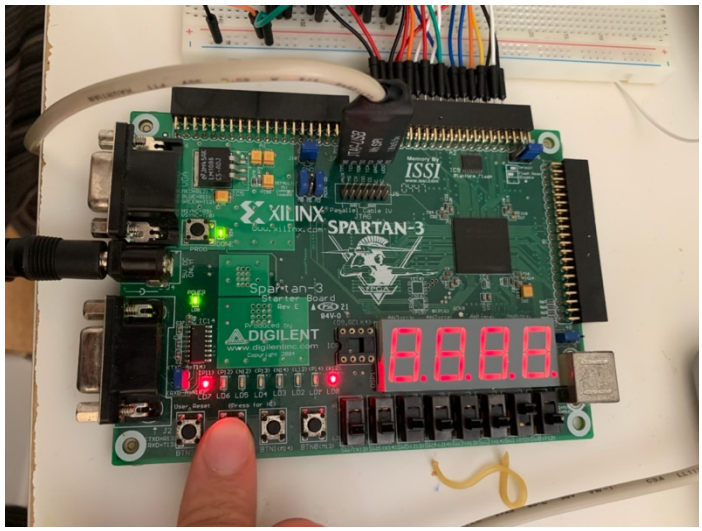


Fig. 15. After Test (LFSR with 8192 tests) – All tests failed (as expected)

6.1.3 Exhaustive Test Equipment Test (Counter)

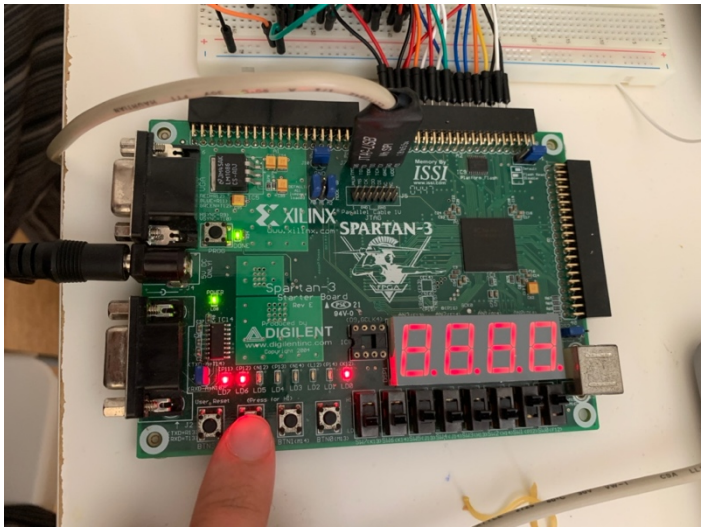


Fig. 16. After Test (Counter with exhaustive test) – Tests failed (as expected)

6.2 Input(1) Stuck At 1

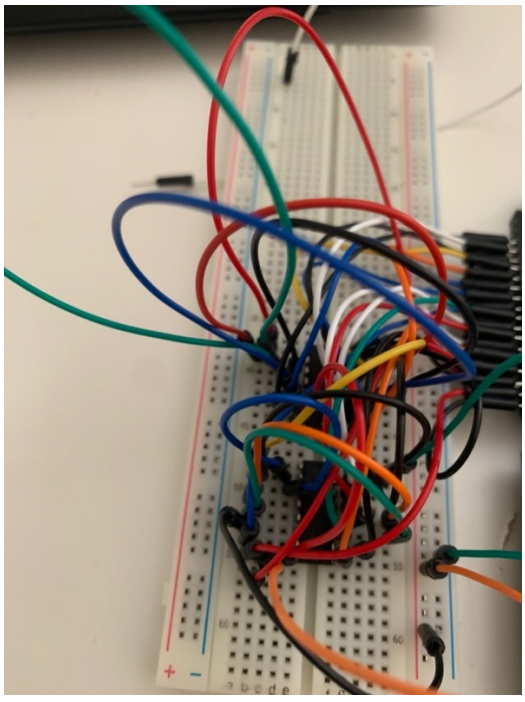


Fig. 17. Inserting stuck at 1 to input(1)

6.2.1 Random Test Equipment 1 with 128 Tests (LFSR)

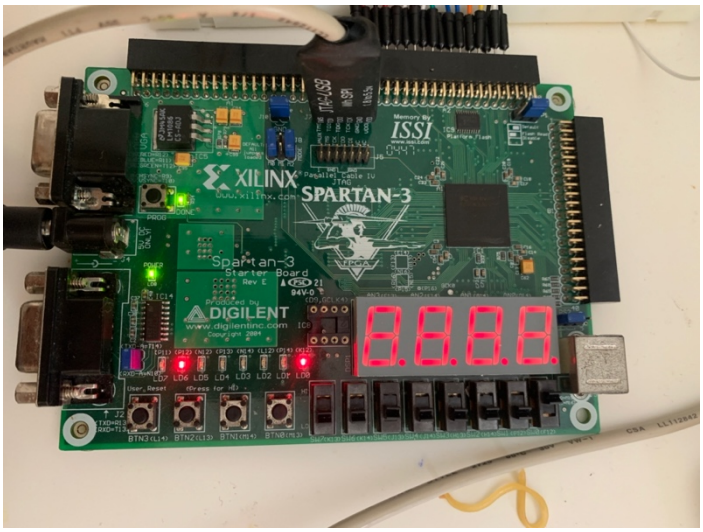


Fig. 18. After Test (LFSR with 128 tests) – All tests failed (as expected)

6.2.2 Random Test Equipment 2 with 8192 Tests (LFSR)

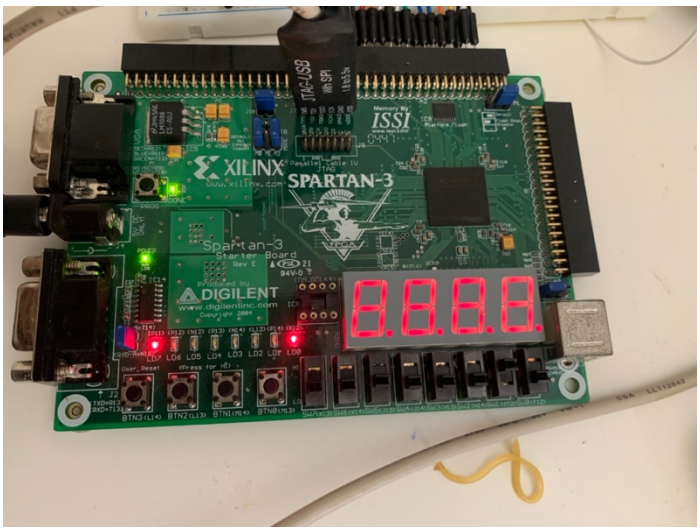


Fig. 19. After Test (LFSR with 8192 tests) – All tests failed (as expected)

6.2.3 Exhaustive Test Equipment Test (Counter)

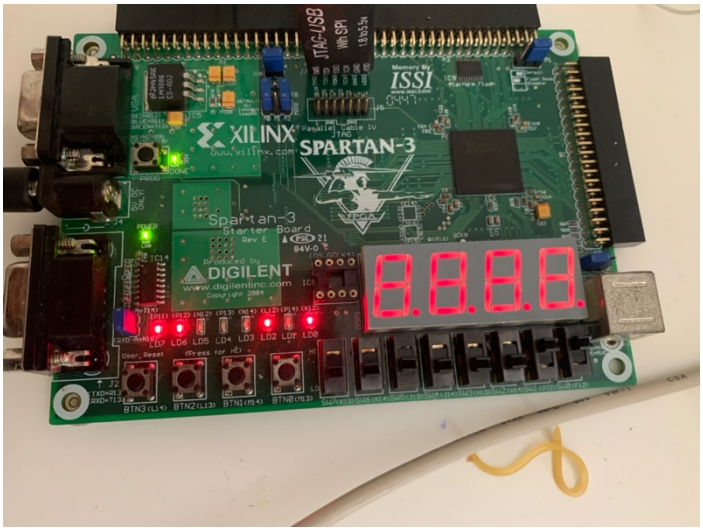


Fig. 20. After Test (Counter with exhaustive test) – Only Signature Generator test failed.

6.3 Input(2) Stuck At 1

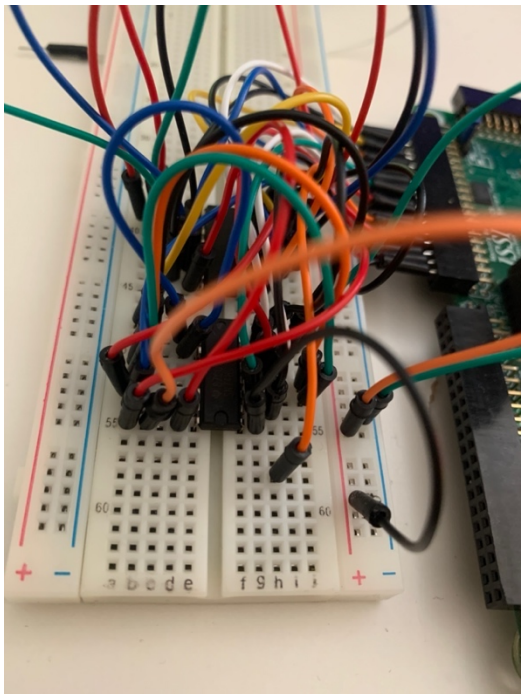


Fig. 21. Inserting stuck at 1 to input(2)

6.3.1 Random Test Equipment 1 with 128 Tests (LFSR)

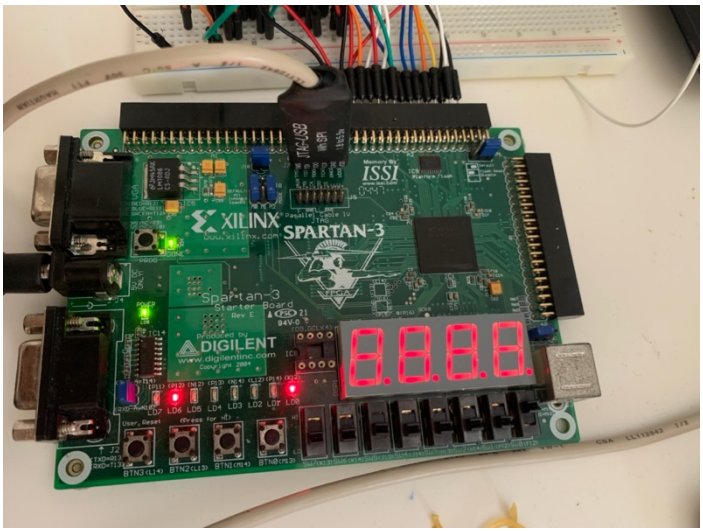


Fig. 22. After Test (LFSR with 128 tests) – All tests failed (as expected)

6.3.2 Random Test Equipment 2 with 8192 Tests (LFSR)

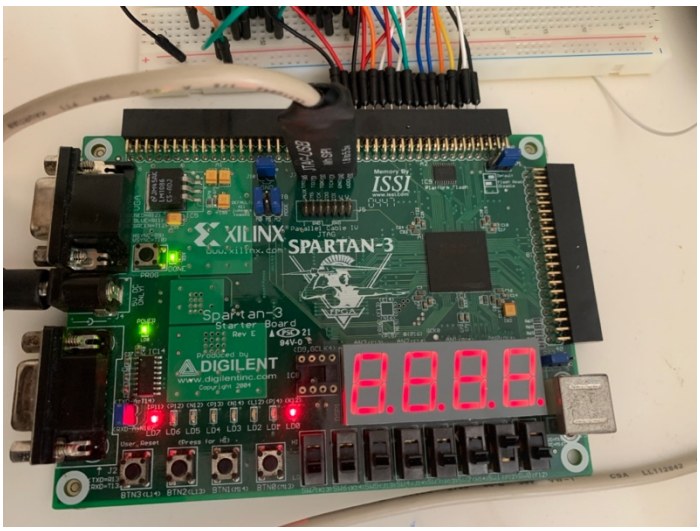


Fig. 23. After Test (LFSR with 8192 tests) – All tests failed (as expected)

6.3.3 Exhaustive Test Equipment Test (Counter)

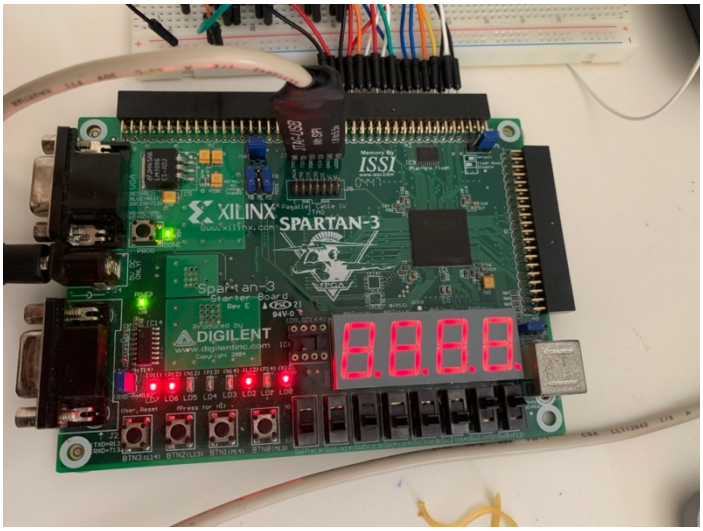


Fig. 24. After Test (Counter with exhaustive test) – Only Signature Generator test failed

6.4 Input(5) Stuck At 0

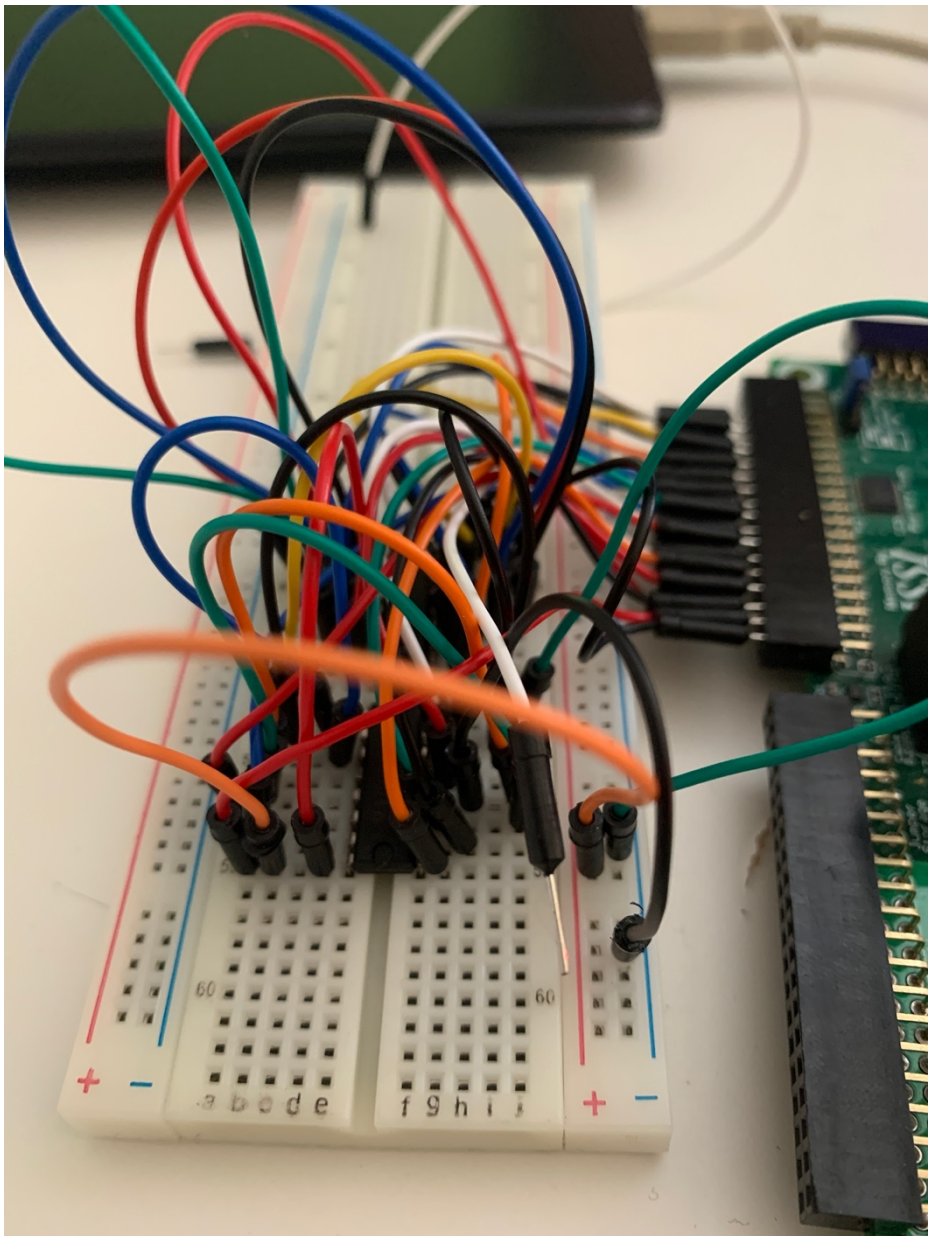


Fig. 25. Inserting stuck at 0 to input(5)

6.4.1 Random Test Equipment 1 with 128 Tests (LFSR)

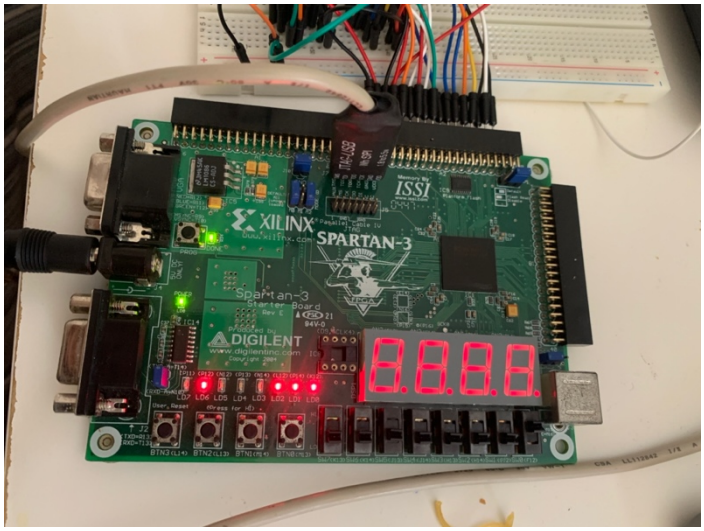


Fig. 26. After Test (LFSR with 128 tests) – All tests passed (Wrong results!)

6.4.2 Random Test Equipment 2 with 8192 Tests (LFSR)

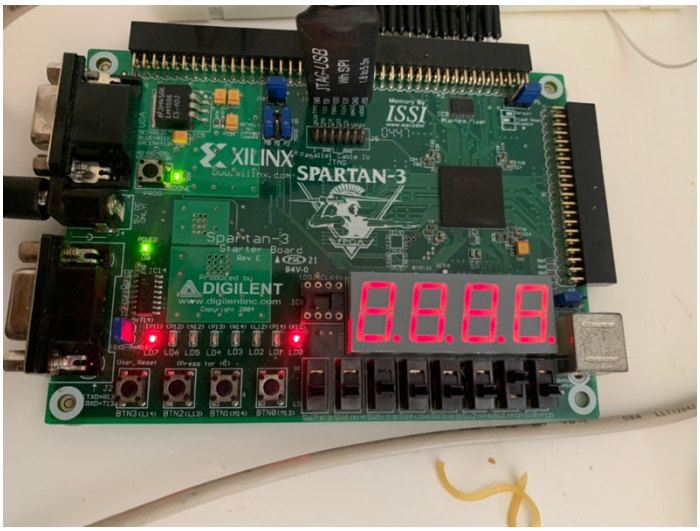


Fig. 27. After Test (LFSR with 8192 tests) – All tests failed (as expected)

6.4.3 Exhaustive Test Equipment Test (Counter)

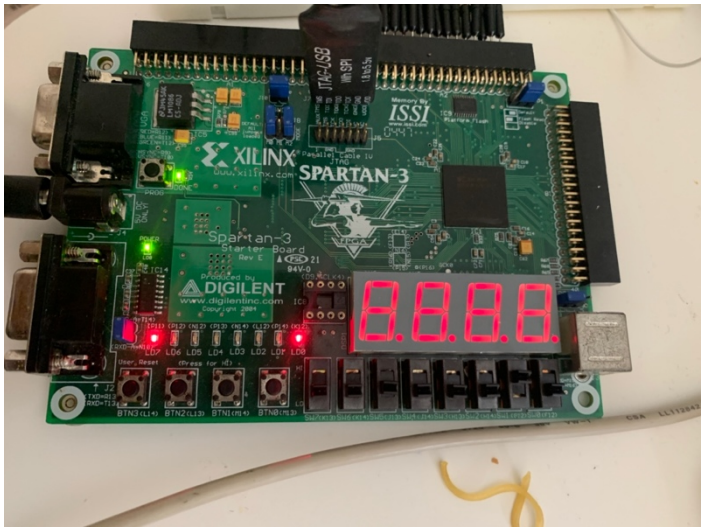


Fig. 28. After Test (Counter with exhaustive test) – All test failed (as expected).

6.5 Input(11) Stuck At 0

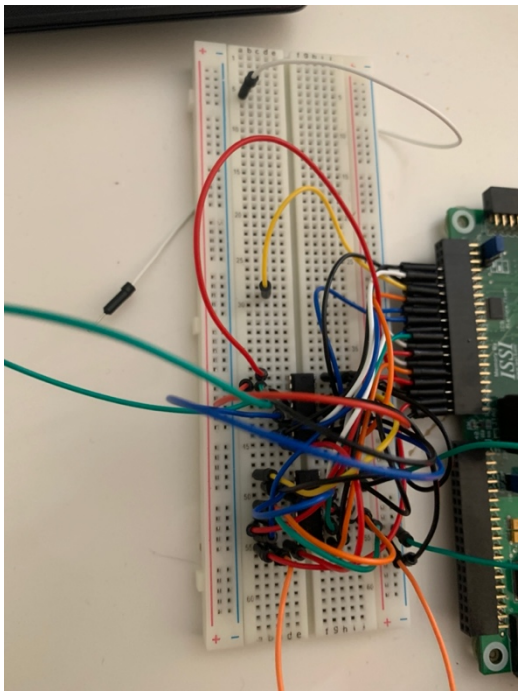


Fig. 29. Inserting stuck at 0 to input(11)

6.5.1 Random Test Equipment 1 with 128 Tests (LFSR)

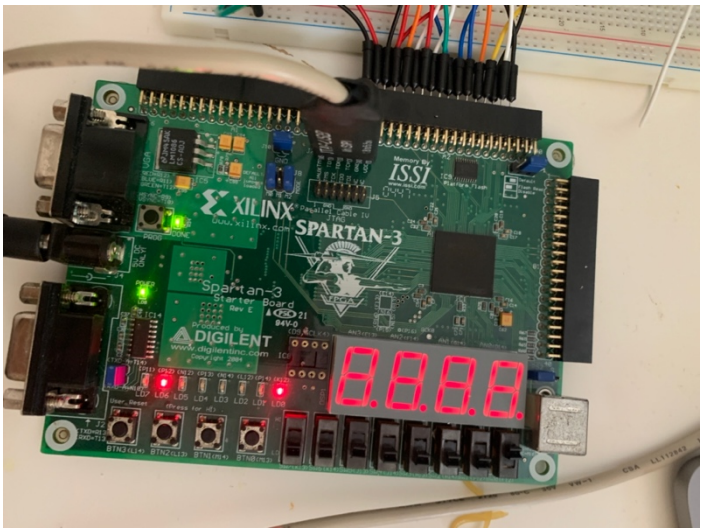


Fig. 30. After Test (LFSR with 128 tests) – All tests failed (as expected)

6.5.2 Random Test Equipment 2 with 8192 Tests (LFSR)

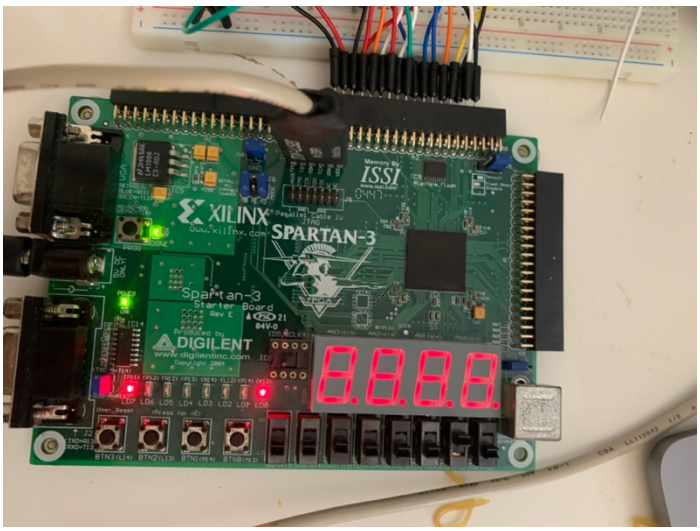


Fig. 31. After Test (LFSR with 8192 tests) – All tests failed (as expected)

6.5.3 Exhaustive Test Equipment Test (Counter)

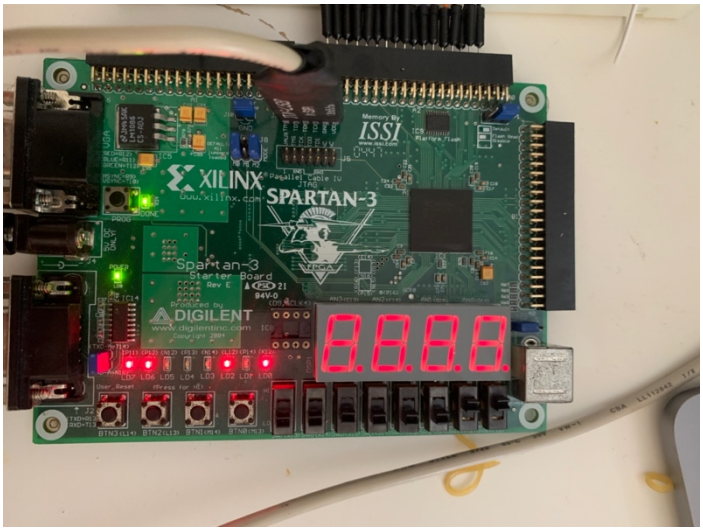


Fig. 32. After Test (Counter with exhaustive test) – Only Signature Generator test failed

We tested 5 failure with our implementation, Input(0) stuck at 0, Input(1) stuck at 1, Input(2) stuck at 1, Input(5) stuck at 0, Input(11) stuck at 0.

For input(0) stuck at 0 failure, all the systems detected failure at the both output of each system (signature generator and compression counter).

For input(1) stuck at 1 failure, Exhaustive test equipment (Counter) did not catch the failure by using compression counter. However, it caught the failure by using signature generator (PASS/FAIL1 output was zero). Other two systems (using LFSR to generate test vectors) found the error by using both compression method.

For input(2) stuck at 1, the results are the same as input(1) stuck at 1. Two systems using LFSR to generate input vector caught the failure by using both compression method. However, Exhaustive test equipment did not find the failure by using compression counter.

For input(5) stuck at 0, random test equipment 1 with 128 tests(LFSR) was not able to catch failure at all. It did not find failure by using both

compression method. Nevertheless, other systems were able to catch error perfectly. This result gives an important conclusion. Test case is a crucial parameter, and it must be greater than hundreds.

For input(11) stuck at 0, both random test equipment using LFSR to generate test vectors were able to detect failure in the circuit by using both compression method. Unfortunately, exhaustive test equipment which uses counter to generate test vectors was not able to detect failure by using compression counter.

7 Conclusion

With the help of executed tests, we were able to learn which methods are more suitable to use in hardware testing. Our conclusion is that using LFSR to generate random input vectors are better than using counter because we almost detected all the errors even with a small number of test vectors (128) generated by LFSR. Random test equipment 2 (LFSR with 8192 test case) was able to detect all the error we had tried.

Random test equipment 1 (LFSR with 128 test case) just missed the 1 failure although compact design. The advantage of it is that it uses less resources than other two systems. Plus, it is the fastest test method we have tried.

About compression techniques, we were surprised to see how signature generator did a perfect job. It would never miss the failure with the efficient number of test case. Also, the probability of aliasing is sufficient even if we use 7-bit long LFSR. Compression counter did the worst job. It missed four failures. Especially, it works awful with the test vector generated by counter because the circuit under test would give 0 or 1 for all the test inputs after some value of counter. Making input test randomized by using LFSR decreases this probability. Hence, LFSR is the best choice to generate input vectors if one uses a compression technique. Even if one wants to implement exhaustive test, one can use LFSR because it will give almost all the inputs except 0. If we consider sequential circuits (consists of memory inside), generating random inputs can have some advantages also.